UNIVERSITY OF TROMSØ UIT

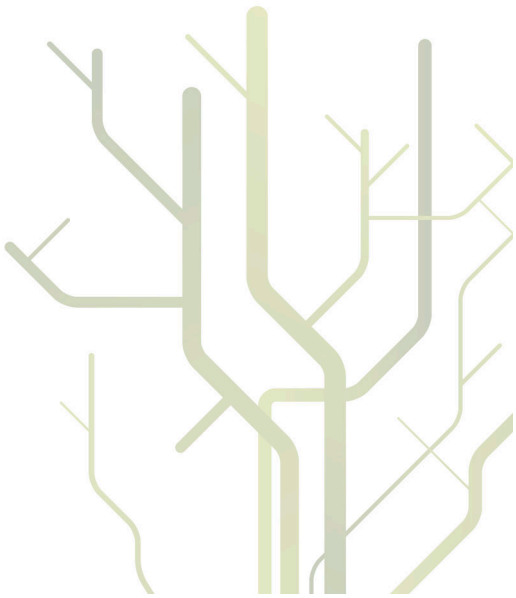
FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

Skynet

A distributed, autonomous filesystem

Svein Ove Aas

Inf-3990
Master's Thesis in Computer Science
May, 2010



Contents

Chapter 1

Introduction

Skynet is an autonomous, distributed, POSIX-like filesystem.

It is distributed: The filesystem can be (and usually is) distributed across multiple physical computers, which each see the same virtual filesystem.

It is autonomous: Files are automatically redistributed between nodes at need, maintaining invariants on redundancy or redistributing free space.

It is POSIX-like: It does not support hardlinks or user/group security, but otherwise follows the POSIX filesystem requirements except when asked not to.

1.1 Problem statement

This section describes the reasoning behind creating this project. To summarize, there is a distinct lack of distributed filesystems that are both easy to use, safe, can store files on multiple computers and can be easily grafted on top of an existing filesystem, or shut down to recover that filesystem. Skynet is an attempt to remedy this.

When picking a network filesystem, the existing ones fall into two main groups.

There are those that are strictly client-server and do not let you spread a single filesystem over multiple machines, such as SSHFS[1] or NFS[2]. These are typically simple to set up, but of course extremely limited in use.

There are also more complex filesystems that are meant to cover multiple nodes - AFS[3], Coda[4] and Ceph[5] just to mention some. These do not share the limitations of single-node filesystems, but as they do not share their simplicity either, they are still somewhat unsatisfactory.

Additionally, some of the most advanced filesystems (Coda being a notable example) require users to create storage partitions using their own spe-

cial on-disk format, which complicates conversion and makes recovery harder due to needing special tools.

The purpose of Skynet, then, was to create a distributed filesystem that lacks those particular weaknesses. This means there were two main goals: The filesystem should use an existing filesystem for storage, as far as possible making this storage tree a partial copy of the distributed filesystem; and, the filesystem should as far as possible take care of its own maintenance.

Although it is impossible for such automated maintenance to reach quite the same level as traditional, human-mediated maintenance, in many use cases it would be good enough. Additionally, this would allow it to be used by nontechnical users who are unable to administrate more traditional distributed filesystems.

Another extremely important goal was that existing programs should keep working - in other words, that it should behave like a POSIX filesystem. Not being an absolute requirement, this target was occasionally missed where examination of usage patterns showed that it wouldn't affect programs, and deviating would simplify the system or improve one of the other targets. There is a long tradition of such adjustments in Unix filesystems, though usually not to this level; to name one, modern Linux kernels mostly disable access-time updates on all filesystems, in direct contradiction of POSIX.

As work progressed, it became obvious that the Haskell ecosystem lacked some fundamental tools for such a system, and so three additional targets were added: The creation of a cryptographically secure message-passing middleware that could provide location transparency, as none existed, and the creation of AES and SHA bindings because the existing ones were unreliable.

1.2 Glossary

Each of these glosses is described in more detail later in the document, but before the main text starts, here is a glossary to refer back to.

Node Any single Skynet node, usually meaning a single machine. Nodes can be moved from machine to machine, or multiple nodes may run on one machine. The latter condition will be assumed not to happen for the purposes of writing clear documentation, but is harmless so long as they do not use the same storage area.

Client A Skynet node currently acting as a client for a user application. This does not refer to the application itself; multiple applications on the same machine will share the same client.

Server A Skynet node with data storage, acting on a request from a client.

Resource A file or directory. Not anything else.

Master A server authoritative for a given resource.

Slave A non-authoritative server for a given resource, acting as backup.

Lease The Skynet equivalent of a file descriptor.

Address Not IP or DNS address, but instead a 128-bit integer designating a Skynet node (via Hermes), which uniquely identifies it regardless of IP changes. As explained in the Hermes chapter, this is actually the hash of an RSA public key.

Administrator A Hermes node responsible for controlling access to the network, by creating cryptographic tokens; does not require an actual trained human administrator. Hermes can be set to a low-security mode where any node can perform this role.

Chapter 2

Skynet

This chapter documents the detailed requirements, design and implementation of the Skynet package, which is to say the actual filesystem in the Skynet project. The middleware and attendant packages are documented in the Hermes chapter.

2.1 Requirements

The problem statement section explains that the most important requirements were, in order of importance, automatic maintenance, simple conversion to/from the filesystem's storage format, and POSIX-compliance.

This section goes into more detail, explaining secondary requirements that are logically required to fulfill the primary ones.

2.1.1 Reliability

In order to prevent any requirement for manual intervention, an important design principle has been that no single or multi-node failure should ever prevent Skynet from completing operations that don't *necessarily* touch the failed nodes. Combined with redundancy in data storage, this makes a highly available, high-performance distributed filesystem possible.

This means that, for some file, if the entire directory structure from root to a requested file is available along with the file itself, then no unrelated node failures or network failures that prevent communication with nodes that store other data should ever affect operations on that file.

Additionally, cached information should be used where possible; for example, once the location of a file is known there should be no requirement to look it up again for later operations. Instead, the authoritative node(s) stor-

ing the file should inform the client if their information is invalidated, once they attempt to use it. This also removes any requirement to communicate such changes to clients.

2.1.2 Redundancy

Skynet offers a master-slave model of storage redundancy. Read performance can be improved by using the slaves as well as the master, but writes can only be streamed to a single node at a time.

This drastically simplifies the implementation (increasing reliability), but may reduce write performance if aggregate bandwidth to the responsible nodes is higher than bandwidth to just the master node. However, such a scenario is unusual; typically the storage nodes will be on the same network, or individually have higher bandwidth available than any given client is likely to have.

2.1.3 Coherency

POSIX semantics require global serialization of all filesystem actions. This is not compatible with a highly available distributed filesystem, where enforcing such serialization would create excessive network traffic at best, and could easily stall it entirely in case of incidental network failures or node failures that can't be immediately characterized as such, even failures of a single unrelated node.

As a compromise, Skynet offers per-file serialization. Assuming POSIX coherency has been selected by all clients involved in a sequence of operations on one file, they will see the same sequence; however, they may see different sequences if comparing different files. This is assumed to be a reasonable trade-off, as very few applications will rely on this POSIX guarantee without using explicit calls to flock.

It is also possible to select the weaker session or eventual coherency when opening a file, in which case all nodes using POSIX coherency still see the same order of operations for all operations involving the file (including those initiated by nodes using eventual coherency), but non-POSIX nodes may not see that order.

For session semantics, clients will never see updates from a different node while holding a file open, nor will other clients see theirs. Implementing this potentially requires extra storage on the file's master for a COW mapping, causing writes by other nodes to block if this space isn't available at the moment.

For eventual semantics, updates are eventually seen by all other (non-session) clients. There is no timeliness or sequencing guarantee on this, but in practice updates will be seen approximately as fast as with POSIX semantics; the difference is that calls involving the file won't block until this time has passed.

The usefulness of this mode is limited, as the performance improvements should only be seen for concurrent access to the same file. It may be useful when different nodes are accessing different parts of the same file, to cover the lack of file segment leases. Implementing such leases would, however, be better, and it is also an extremely unusual mode of operation.

2.1.4 Security

The only security currently offered is the ability to limit network access to trusted nodes. Only the administrator node has the ability to grant access to a Skynet network; without a signed cryptographic token from this node, it should be impossible for third parties to break in.

In the future, it would be possible to implement the POSIX user/group security model by adding per-user/per-group cryptographic tokens. Properly integrating this with the underlying OS of each node where it is mounted is a bit of a research topic, however, as user ID numbers can differ between nodes and it is not entirely obvious what to do if the system requests access to a file it does not have access to.

Denying the request is the obvious choice, but many programs appear to assume that if the user IDs of the file and caller match, they have access. Breaking this assumption on several occasions caused the GNOME desktop environment to crash the moment the filesystem was mounted, which is not a desirable feature.

2.2 Design

Skynet is roughly divided in five sections.

There is a middleware layer - Hermes - which handles communication, authentication and encryption, and hides the physical addresses of Hermes nodes behind a unique identifier. "Address" refers to that identifier in the rest of this document. It is described at length in the Hermes chapter.

The storage module abstracts disk storage, caching and journalling or using COW capabilities in the underlying filesystem transparently. It is conceptually divided into directory and file storage, though nodes can and do offer both. This layer is provided as a Haskell API, and does not directly

touch any network interfaces or Hermes; it is thus limited to single nodes, though the interface is tailored to implementing the distribution layer.

The server module mediates inter-node communication, handling normal file/directory operations and replica maintenance and creation as well as crash recovery, but does not decide when to do any of this in the first place; it provides mechanism, not policy.

Of the remaining sections, one is a FUSE[1] interface that provides normal applications with access to the filesystem. The other is a set of autonomic management routines that can ideally replace a human administrator, which enforces user requests such as a given level of redundancy and performance by moving replicas around.

These management routines may in extreme circumstances such as disk failures that causes unrecoverable loss of redundancy request assistance from a human administrator, but should be capable of handling all routine maintenance themselves.

2.2.1 Storage layer

The storage layer handles all persistence in Skynet, which is to say all state-changing operations that require crash-proofing go through this layer. This includes file and directory operations, but also maintaining replica pointers and leases.

It currently uses haddock-state for all storage except bulk file data and some file metadata. Since haddock-state is an in-memory persistence layer, per-node filesystem size is currently limited by the size of their memory.

The reason for this is that, while it is easy to find persistence layers that provide ACID, systems that provide a subset of ACID - in particular atomicity without durability, or reporting when an operation is complete but not blocking - are relatively unknown, yet vital for filesystem performance.

An experiment in creating one was carried out.

Btrfs

The as-yet incomplete Btrfs[9, 10] filesystem exposes low-level transaction primitives that could be used to build such an improved system. Explaining how they work is beyond the scope of this document, but there are four main primitives of interest, implemented via IOCTLs:

Transaction start/stop Btrfs is a transactional filesystem. It normally creates transactions as it sees fit, but by using the TRANS_START/TRANS_END

[1] Filesystem in USErspace

ioctls, it is possible to manually invoke a transaction. Transactions are ordered, but - importantly - not durable. Calling TRANS_END does not guarantee that the transaction has been forced out to disk, it merely tells the filesystem it is now allowed to do so. (Data may be written out during a transaction, but will not be visible until some point after calling TRANS_END).

As most POSIX file operations fall into this same "atomic, ordered, but not durable" category, having the ability to create such transactions is an absolute requirement for achieving high performance.

Some caveats apply, however. Most importantly, it is relatively easy to crash the filesystem by using these ioctls; many operations that touch the metadata, even when that is not obvious, can only be executed outside of a transaction. This includes such operations as allocating more space for a file.

Further, using the transaction calls would prevent any other programs (than the Skynet daemon) from using the filesystem, necessitating that it is hosted on a separate partition.

The transactions also are not atomic, which necessitates a certain amount of cleverness to build atomic primitives on top of them.

Sync Occasionally, you *do* want the transaction to be durable. Calling sync is the way to do this; it is not significantly different in effect from fsync, but affects the entire filesystem instead of just a single file.

Clone/clone_range Btrfs is a copy-on-write filesystem. This also means that multiple files can share the same data blocks; by using the clone ioctl, a file can be very quickly copied, which would simplify the implementation of session semantics dramatically.

Using the current Linux Btrfs driver, an attempt was made to exploit these primitives. This had tragic consequences, however: A misunderstanding of what operations were safe inside a transaction caused a filesystem crash. Worse, a crash that silently corrupted recently written files, which did not become obvious until much later.

Given the apparent number of sharp corners left in the implementation, therefore, it is preferable that it be left to mature for another year before attempting such a thing again.

API / Semantics

This subsection describes the storage module interface at a high level. For call details, see the generated Haddock documentation. Where not otherwise specified, POSIX semantics apply.

For all data stored by the storage layer, in addition to the metadata mandated by POSIX there is an additional tag specifying whether this node is authoritative for the data, as well as a list of other servers containing the data (with the master marked).

It must be noted that this metadata is not maintained by the storage module itself; it is the responsibility of the server module to keep it up to date.

The directory API is mainly identical to the POSIX directory API, with a few alterations:

- File references, instead of the traditional i-node integers, are an URI-like structure usable for identifying files across nodes. File references consist of the address of its master, as well as its path. The choice of using a path instead of indirection via arbitrary i-node integer dramatically complicates the implementation of rename(), but is nevertheless desirable. In order to simplify failure recovery, specifically the ability to shut down Skynet and recover the filesystem from its underlying storage, there needs to be single canonical pathname per file anyway; the rename() complications would exist regardless of i-nodes, so long as such an association exist.

- Hardlinks are not allowed. The main reason to not allow hardlinks is that they, while technically possible, are very badly supported on Windows and it is desirable to maintain at least theoretical portability. Additionally it would complicate failure recovery since, again, there would not necessarily be a single canonical name for each file.

- While the Windows complication also applies to symlinks, the cost of supporting them is close to zero. They will therefore be legal until such a time as Skynet is ported to windows, at which point a decision on how to handle them must be made.

- The file/directory APIs do not have a notion of file descriptors as such, but uses an expanded notion called leases. These have several properties apart from the simple fact of representing a handle on an open file or directory:

  - A lease is either local or proxy. Proxy leases amount to temporary assignments of authority to other nodes, allowing for local write-buffering or caching while maintaining coherency; local leases are merely an indication that some client has the file open.

- Each lease has an associated timeout. Leases can be refreshed an arbitrary number of times, but if it isn't refreshed before it times out, it is automatically removed and any resources held (such as otherwise deleted files) are removed.

- Leases can be read-only or read-write; in this case POSIX has more modes. All the POSIX modes other than read-only map to read-write; in particular, POSIX append mode alters the write command offset, not the lease. This is necessary, as multiple POSIX file descriptors (from multiple applications on the same client) can correspond to one lease. A lease can be promoted from read-only to read-write, or demoted back down, the former operation may block (in the server module) due to coherency requirements.

- A lease can require one of three levels of coherency:

  POSIX All clients which have the file open in this mode will see the same order of updates.

  There can be at most one read-write proxy lease of this kind, and any number of read-only proxy leases so long as there are no read-write POSIX coherency leases of any kind. Any number of local leases are allowed; therefore, if multiple POSIX leases where at least one is read-write are simultaneously required, existing proxy leases should be invalidated and made local.

  Session Any client which has the file open will see no updates by different clients until it is closed. This may require duplicating the file on the server, which depending on available space can cause whichever operation required the duplication to block or trigger re-balancing.

  Unsynchronized/Eventual Anything goes. Clients must explicitly synchronize access patterns between themselves and flush as necessary.

  Note that coherency limitations on buffering/caching are merely advisory at this level. There is no way to prevent a client from caching data inappropriately; the API limitations are an aid to correctness, not a guarantee of it.

Data Model

This section describes how data is physically stored, in terms of the operating system(s) Skynet is built on. That is to say, the physical layout of a single

node; not that of the virtual filesystem, which is covered in the Distribution Layer subsection.

The format is designed to be accessible without special tools, and to be easy to repair or convert to/from a normal filesystem.

All descriptions are relative to a root directory, configured by the user.

Files are stored in the files/ subdirectory, exactly as they appear in the filesystem (modulo write buffering).

In practice, you can recover a working monolithic filesystem from Skynet by unioning all the files/ directories on the various nodes, preferring master nodes over slaves where possible since slave nodes may contain outdated data. Permission bits on the directories may be lost, however.

Directories are stored using happstack-state, which persists to the meta/ subdirectory. A directory consists solely of a list of file or directory names, along with the directory permission and time bits; information that exists per-file in POSIX filesystems is stored in the file, not here.

Skynet-specific metadata such as master/slave node relationships or lease states is also stored using happstack-state, which uses the meta/ subdirectory for all its data.

2.2.2 Distribution layer

Apart from the POSIX-mandated file/directory data and metadata which is kept on the master node and replicated on slaves, there are several bits of extra metadata per resource which must be maintained across nodes, as follows:

The master node - that is, the node which at any given time controls the resource.

The coordinator - The master of the parent directory of a given file or directory. To clarify, for a file /a/b/c the coordinator is the master of b, not a or c.

The coordinator is critical for failure recovery, but is not otherwise involved in normal operation.

Slave nodes - the master node's backups, which are kept continually updated.

Lease nodes - which nodes which currently hold leases on the file/directory.

As a special case, the root directory has no coordinator.

The canonical record of all these facts is held by the master node, and updated on the other nodes involved as described below.

Normal operation

To reduce latency, a modified RPC mechanism is used. For all queries that could touch multiple nodes, the full details of the query is passed to the first node queried, which then sends it on to the next appropriate node instead of using a round-trip through the client.

Additionally, to mitigate the effect of lost messages, partial information sufficient to skip nodes up until the next hop (thus, containing at the very least the address of said next hop) is passed back to the originating node at each hop.

In all cases, a configurable number of timeouts of configurable length are allowed before any failure mode is entered. However, multi-hop queries never trigger either retries or timeouts in nodes beyond the originating client; in fact, information is rarely passed backwards through the chain, but always directly to that originating node.

Most file and directory operations do not touch multiple nodes, however; only the client node and the master and/or slave nodes for that particular directory or file. While other client nodes are involved inasmuch as there can be contention, there is no direct cross-client talk; contention is handled through locking on the server nodes, by acquiring leases on the files or directories.

Locating directories and files A recursive lookup is used. For a file /1/2/3/4:

A cold client sends the full path of the directory to the master of the root directory; this node resolves the path as far as possible, then passes the message on to the master node responsible for the next directory in. Ownership information for the directory chain is passed back to the originating client as possible.

This continues until the master node for the requested directory/file has been located. This node is not contacted in the process, except if it's also the master node for the parent directory of the requested resource.

Returned information is cached by the client, so later lookups can start deeper in the tree. If such a warm client sends the query to the wrong node for 4 (probably due to relocation, or error recovery), this node passes the query on to the deepest node it believes it knows the location of (3, 2, 1 or

even the root); if this information is wrong, this procedure may also recurse. Cache invalidation messages are sent backwards through the chain, though one hop only.

Addresses are aggressively cached on the client, as they rarely change. If they turn out to be mistaken, requests to a node that does not own the file/directory will simply return an error, allowing for invalidation and a new query.

Leases Location operations can get away without locking because outdated location information only triggers a re-query. All other operations (described below) require some form of locking, however; mostly, acquiring a lease on the file. These leases also double as file descriptors, as described in section 2.2.1.

Several POSIX directory operations do not involve file descriptors. These have been altered to use file (directory) descriptors that double as locks on the server, called leases, to allow higher performance. The FUSE interface hides this from user applications.

Lease requirements are described for each individual operation. A lease is requested using the Open RPC call, which does not need to correspond to open/opendir system calls by the user; for example, the rename() call will first require opening both the source and destination directories.

In case of write conflicts, where POSIX level coherency is requested, Skynet resorts to a "Local" lease mode where calls are unbuffered and uncached - using the master node as the arbitrator of call order. If a file/directory is opened that is already held open by another client, depending on coherency requirements and mode the open call may block until said other clients have either had their lease demoted to Local or have timed out.

Any operation that requires a read-only lease will also work fine with a read-write lease.

All directory operations use POSIX coherency, regardless of configuration.

Leases are always managed by the master node of the file/directory in question. The lease comes with a list of up-to-date slave nodes, however; read operations can thereby be distributed across the slave nodes.

Lease acquirement/releases are mirrored to the slaves, so if the master fails a minimal number of leases will be lost. However, the master does not wait for acknowledgment from slaves before replying to lease requests, which leaves open the possibility that a lease may be stolen if a master crashes. This is a performance trade-off that will in the future be configurable.

open/read/write/close are implemented in the most direct possible fashion. open acquires a file lease, read-write iff write mode is requested, otherwise read-only. read, write and close simply use this lease.

As one lease can correspond to multiple file descriptors on the client side, there is also a "reopen" call to turn a lease from read-only to read-write or vice versa.

Write calls return as soon as the file has been written to the master node's filesystem, and does not wait for slave nodes. This will in the future be configurable.

Write calls are only valid on the master node, while read calls are valid on any relevant node. Slave nodes of a file inform the client when a lease has been mirrored to them and they are available for reads.

The coherency parameter is set by per-node user configuration.

fsync explicitly flushes the client-side write buffer of a file or directory, and blocks until it has been stored on at least the file's master node. This may take a long time. It does not require any particular kind of lease, though using it on a read-only lease will tend to be a no-op.

truncate, utime require a read-write lease on the file.

fstat,access,readlink require a read-only lease on the file/directory.

opendir requests a read-only lease. Further directory operations may be able to use this lease, or may request that it be promoted first. While opendir does not actually require a lease of any kind, it nevertheless blocks until the lease has been acquired.

readdir requires a read-only lease.

closedir requires a lease, specifically the one returned from opendir. The lease is not immediately released, as there is a good chance other operations using the directory will follow.

mknod is not supported, and will return EIO. Device nodes on a network filesystem are an obvious security problem; it may later prove useful to optionally allow this, but for the time being there's no need to tempt fate.

mkdir, unlink, rmdir, symlink, chmod, chown require a read-write lease on the parent directory.

link immediately returns EMLINK. See section 2.2.1 for a discussion of why hardlinks are not allowed.

rename is the most complex call, as it frequently touches multiple nodes. Renaming a file requires read-write leases on the file and both directories involved, as all are modified; the directories to remove/add a file entry, the file to change its path.

Two-phase commit is used for this, with the client doing the rename as the coordinator. As leases already have timeouts, if the client fails during the transaction the leases will eventually time out and the transaction can be rolled back, without further issue.

Renaming a directory is a more problematic matter. In this case, any number of sub-files and subdirectories will have to be recursively modified, which could cause the filesystem to grind to a halt attempting to lock all of them. In lieu of a better solution, a rename of a directory will return EXDEV - suggesting that the source and target are on separate filesystems, even if that is not the case - which will cause the calling process to use a copy-delete algorithm instead.

This works, but is very much not desirable. A possible improvement would be to use the file-renaming algorithm recursively, moving files one at a time instead of all in one transaction. This would however expose an intermediate state where the renaming is partially complete. As this is essentially the algorithm all known file-movers fall back to if rename returns EXDEV, this should not be an issue, but more research is required to make sure.

2.2.3 Failure handling and recovery

If a slave node is unreachable within a reasonable amount of time (picked to limit network traffic and false timeouts; any value will allow correctness), it is considered to be down. No action is necessarily taken other than blacklisting the node for a period of time so it will not be contacted again (by this particular node), so long as there are others that can answer queries.

If a master node discovers one of its slaves to be down, this can trigger a maintenance routine to look for a new slave. To avoid excessive network traffic, the minimum and maximum number of slaves should not be set equal.

If a master node is unreachable within the same reasonable amount of time, it is considered to be down. In this case, other nodes may take contingency actions as follows:

1. Any nodes which require access to the downed node to reach a particular resource may appeal to the coordinator for this resource. If the coordinator (and thus the parent directory) is also down, the procedure recurses, attempting to recover the parent directory in turn. If the root directory's master is down, the system will freeze until it is back up; please try to ensure this one node is reliable.

2. If the appeal applies to a node that is not the master for the resource, presumably because it has just been replaced through a different appeal, a pointer to the new master is simply returned. Otherwise:

3. The coordinator picks a random slave to serve as the new master. The slave is informed of its new responsibilities; if it happens to be down as well, the coordinator repeats until it finds a working slave. If all slaves are down, it gives up and reports failure to the appellant; otherwise, once a functioning slave has been promoted, success is reported.

The root node is an obvious weakness in this algorithm. One possible solution would be to have the root directory slaves regularly ping the master to ensure it is up, and initiate an election if it fails. This has however not been implemented.

Another weakness is the fact that if a master fails and then comes back up, it will believe it is still the master. Given that addresses are aggressively cached by clients, it is then possible for a client to contact it believing it is the master, despite the parent directory's master (that is, its coordinator) knowing better. This is impossible to fully prevent while maintaining function in the face of network partitioning, but a limited attempt is made: The new master will regularly ping the old one, trying to tell it it is no longer the master. Additionally, once the old master eventually contacts a slave to attempt a write, the situation will be explained.

If, perhaps due to actual network partitioning, we still end up with two masters being written to, once the situation is eventually discovered the master with the newest modification time for this file will retain its status, while the old one discards its copy. This is the sole case in which data loss is possible.

If the resource is a directory, it is unioned instead - hopefully preventing significant data loss, but perhaps causing undeletion of files instead.

2.2.4 Client layer

This section describes the interface between FUSE and the distribution layer.

Overall, the client layer has four components:

The lease refresher maintains active leases by periodically requesting their renewal from their master server.

The buffer buffers all updates to leases that are held in proxy mode, sending them in FIFO order one at on a per-server basis; one send can be active at a time per server.

There is a configurable maximum buffer size, measured in bytes.

The cache maintains copies of all cache-able data, defined as all data returned from queries on a lease in proxy mode. There is no size limit on this, although the data is removed when a lease is closed or put in local mode.

The FUSE interface mediates between FUSE and the distribution layer.

FUSE methods match relatively directly to distribution layer messages. In general, they are implemented as follows:

1. Resolve the path of the resource in question. These are always cached, so this is typically cheap.

2. Ask for a lease. If we already have one with the appropriate mode (read-only/read-write), use that; otherwise, reopen an existing one (if there's a mode mismatch) or open a new lease.

   If this operation fails, either due to a timeout or an error, the client will first attempt to resolve the path from scratch before possibly invoking node failure recovery.

3. For updates: if the lease is in proxy mode and there's space in the buffer, stash the call in the buffer, otherwise block while calling the server.

   For both queries and updates, the changed/fetched information is stored in a local cache if the lease is in proxy mode, otherwise discarded.

If either of the last two operations fail due to a timeout, this will cause the downed-node failure handling to be invoked for this node. Any other server-side error is passed on to FUSE.

There is one exception to this rule. The flush (fsync) call is implemented by extracting all updates to the given lease from the buffer, then blocking until they have been completed.

2.2.5 Autonomic maintenance

Rebalancing has three purposes: Handling intra-node out of space conditions, ensuring a configurable level of filesystem redundancy, and optimizing performance.

Of these three, only the first two are handled autonomously. Performance is mainly ensured by aggressive caching and buffering, with any performance-related rebalancing being initiated by the filesystem user.

Each node broadcasts its free/total space statistics, along with some basic notion of reliability (you wouldn't want to use a laptop as a master node) via the gossip functionality of Hermes.

Reliability is currently a boolean value - one or zero - but is implemented as a 32-bit integer to leave space for cleverer implementations in the future.

Out of space conditions When a file grows, it is possible for it to outgrow the available space on a node. If the node in question is a slave, the master node is informed so it can pick an alternative slave node, and the file then removed once the master agrees.

If the current node *is* the master node, it will attempt to transfer its master rights to one of its slaves first, to avoid stalling the writer(s).

To avoid a hot potato scenario where the file ends up losing all redundancy, below a minimum number of replicas the master will stall writes instead of allowing replicas to be removed. Write operations can resume once the number of slaves is again above this threshold.

Creating replicas If the number of slaves for a master is below the configured minimum limit, it will look for a reliable node with sufficient space to hold the entire file or directory, then contact the node and allocate space. (If the node is down, naturally, trying another until it finds one that works.)

Filling works differently depending on whether the resource in question is a file or a directory.

For directories, the space is allocated and contents transferred in a single, atomic transaction.

For files, the space is allocated separately from data transfer. After allocation, the master will stream the data over in bite-sized chunks[2], maintaining a pointer of how far it has gotten. If the file is written to at a point before the pointer while this process is in progress, the writes are immediately streamed before the process continues.

[2]Meaning, at the moment, 64 kilobytes

After streaming is complete (or immediately after creation, for directories), the coordinator node is informed of the new slave and it becomes eligible for read access by clients.

When the minimum and maximum number of replicas aren't equal, which they shouldn't be, once replica creation has started it will repeat until the maximum number has been reached.

Chapter 3

Hermes

Hermes is a self-maintaining peer-to-peer messaging system, which provides distribution transparency in the form of hiding the physical addresses and address changes from the library user.

Instead, each node is identified by a logical address, the hash of their RSA public key.

It was designed for the purpose of supporting a high-performance low-node-count distributed filesystem, and is therefore optimized for performance in small networks rather than scalability, for reasons given on page **??**.

In particular, each Hermes node maintains addressing information for every other node in the network, and per-node overhead (bandwidth and storage) is generally proportional to the size of the network.

There are two modes of operation - unicast and gossip. Unicast provides a best-effort, fail-fast mode of communication, while gossip provides a tuplespace-like abstraction without any guarantees on ordering or speed, but tolerant of network failures. Additionally, an RPC library built on unicast is provided, sharing its limitations.

All messages use public-key encryption to ensure confidentiality and authenticity, with symmetric session keys as an optimization.

Haddock documentation for Hermes can be found at `http://hackage.haskell.org/package/Hermes`.

3.1 Network requirements

Hermes works as a single-layer peer-to-peer network, which requires all nodes to be able to talk to all other nodes. It makes no attempt at traversing firewalls, instead providing IPv6 support for the purpose.

A node can be configured with a different public and local IP, which allows port forwarding to be used.

However, at the time of writing we're less than two years from IANA IPv4 address exhaustion. As NAT solutions have gotten more baroque over the years, and now often use multiple levels of translation, it would be unwise to rely on port forwarding to work in the long term.

3.2 Modules

Hermes is divided into a number of modules, most representing different modes of communication.

While they are mostly distinct, all modules have at least three functions in common: One to create a new module context, one to snapshot it for storage to disk, and one to restore it.[1]

In one case (RPC), snapshots contain no actual information. However, the snapshot functions are still provided to ensure forwards-compatibility of your programs, as there is no guarantee that Hermes won't change in the future to add information to those snapshots.

Additionally, a simplified wrapper module (Network.Hermes) is provided, which should be sufficient for most purposes and will have a greater emphasis on interface stability than the internal modules. For purposes of versioning, only changes to this module will be counted.

This wrapper interface is essentially the the union of the below described modules, and so is not separately described.

3.2.1 Core (unicast)

The low-level Core module handles protocol setup, encryption and authentication as well as unicast messaging and connection and address management.

Unicast messages are keyed by the binary serialization of a tag, as well as the type of the tag and message. There are separate untagged send/receive functions, which do not interfere with the tagged ones. (Internally, a special "NoTag" tag type is used.)

When receiving unicast messages, in order to avoid queuing messages potentially forever, the receiver must explicitly request the delivery of particular tag/type combinations. This is done automatically by the receive call, but may also be done explicitly. It is expected that most applications will do this on startup, before activating listeners.

[1] In the future, the snapshot functions will be obsoleted by use of happstack-state.

Finally, if a message is rejected in this way the intended recipient will attempt to send a RejectedMessage in return, which includes the tag and original message type, but not the message contents. This is naturally not recursive; a RejectedMessage that is rejected is dropped instead of bouncing.

Future

While the core works well as it is, it is hard to read and could benefit from a refactoring. In particular, the address book should be separated out.

Also, the tags could be deserialized and compared using Eq/Ord instead of being compared in binary form.

For many applications, it would be convenient to turn off the rejection logic. Although it is essential to avoid memory leaks in complex applications, simple ones have no need for it.

Reliability

Since there is no real way to guarantee that messages are received *and acted on*[2] or even that errors are detected, and trying would drastically reduce performance, Hermes does not try.

Instead, error detection is provided on a best-effort basis. In practice, this means that any errors in the connection setup phase are converted into exceptions, including if a remote host unexpectedly closes the connection and Hermes is unable to reopen it; almost always, if no exception is thrown the message will have been successfully received, but it may have been dropped after sending - either by the network or receiver.

In particular, messages are sent at most once. If you receive an exception, the message has not been sent; if you do not, then it has been sent but may not have been received.

If message acknowledgment is desired, feel free to use the RPC module with a ()-typed return value. However, keep in mind that this is not the same as actual transactions, which should be implemented by the library user if needed.

Performance

Hermes is thread-safe[3], but not inherently multithreaded, and will serialize send calls single hosts. Only when sending multiple messages to *different*

[2]Ref: The end-to-end argument

[3]Hopefully. If you have strange problems in a multithreaded program, there may well be undiscovered bugs.

hosts can it leverage multiple CPU cores for processing and encryption, or send in parallel; on the receiving side, only when receiving messages on multiple listeners, although deserialization always takes place in the thread that calls recv.

This is not an inherent flaw in the design, but merely an implementation detail, and could be fixed in future iterations.

Security

As Core is the base module, its security model is largely what the other modules use as well.

Messaging requires any node you communicate with to have an RSA key for authentication. The AES session keys used for communication are encrypted using these keys; only negotiation details such as the HermesIDs of each host is transmitted unencrypted.

Additionally, Hermes requires some assurance that the keys provided belong to someone trusted. Depending on the trust level set, the proof required may be none at all (for test networks), a signature from a trusted authority (see the signature functions in this module), or even that the key is explicitly added by the Hermes library user.

Both the RSA key of the sender and the receiver are checked in this manner, before any information is transferred.

3.2.2 Signature

The signature module allows you to create authorities, and use them to sign a node's keys. It is possible to use this in the traditional manner, creating a signature request on the node, having the authority sign it and then returning it.

However, there's also a convenient function in the Network.Hermes wrapper module which allows you to create a context with its key pre-signed by the authority, and that authority inserted as authoritative.

You can then serialize the context, and move it to the machine you want to use it on.

3.2.3 Gossip

The Gossip module allows a form of blackboard communication. Each node may insert an arbitrary number of "factoids" in the network, which will be spread through the network using a standard gossip protocol.

Eventually, barring network partitions, the inserted factoid can be looked up by key and/or originating node on any other node in the network.

Similarly to messages, factoids are keyed by type, tag and tag type; only matching factoids will be returned. They are also keyed by the inserting node; when requesting a factoid, this key can be left as a wildcard, in which case all matching factoids are returned.

It is also possible to tag factoids with a timeout. In this case the network will keep track of how long it has been since a factoid was inserted and will attempt to drop it once past the timeout, providing a simple form of garbage-collection. This function is not meant to be absolutely reliable, particularly in the face of deliberate attacks.

Performance and Reliability

As factoids are transmitted via a gossip protocol, unlike unicast messages they will in practice always arrive, barring network partitions. There is no absolute guarantee, however; sufficiently bad luck could theoretically prevent them from ever arriving.

However, using this protocol induces a per-node load on the network proportional to the size and number of factoids in the network. Additionally, factoid transmission takes time; although it is possible to reduce this time by reducing the gossip interval, network load is also proportional to gossip frequency.

When a factoid is first inserted, an dissemination mode is triggered that will, with a high probability, transmit it to a large proportion of the nodes on the network. This drastically reduces the time required for full dissemination, as well as quickly preventing factoid loss unless a large proportion of the nodes are simultaneously lost.

The dissemination mode will attempt to transmit the factoid to every node the disseminating node knows of. This procedure works recursively; a node contacted in this manner will itself start attempting to disseminate the factoid. It stops once a node that already knows of the factoid is contacted, or in the unlikely event that every node is contacted.

To limit network load while avoiding stalls from unresponsive nodes, a maximum of one node is contacted every half second, but any number of nodes can be simultaneously contacted if contact with one takes over half a second.

Security

The factoids are signed by the originating node, to prevent the possibility of fake facts. However, it is still possible for a node to insert an arbitrary number of factoids, thus arbitrarily increasing network load.

One way of preventing this would be to add a max per-node factoid count, but this is not currently implemented. Another would be to revoke their access, but revocation certificates are not currently implemented either.

Additionally, a node could wrongly reset the timeout values for arbitrary factoids, causing them to hang around in the network forever.

This could be fixed by tagging the factoids with a signed time-of-insertion at the originating node instead of using intervals, at the cost of requiring all non-compromised nodes to have synchronized clocks. This would be a cure worse than the disease.

As there is no significant security risk in having factoids hang around too long, and purely malevolent attackers are assumed to be rare in practice, the problem has not been further examined.

Assumptions

The gossip module assumes that the system clock on any particular node is monotonic. If this assumption is broken, any factoids with timeouts set may be removed either before or after they should be; however, a single change is unlikely to cause problems, as any lost factoids will be recovered from some neighboring node at the next gossip interval.

More problematically, in the event that a clock is reset to an earlier time, an older factoid (timestamped after the current time) may override a newer one. It is therefore recommended to use some variant of NTPd in to avoid this, as it can perform backwards corrections through skewing the clock instead of stepping it.

There are no assumptions about clocks on other nodes, other than that those should also be reasonably monotonic.

Future

A Gossip variant that uses a central server (or multiple servers) instead of a peer-to-peer protocol would be a useful addition, in the common case where there are at least *some* reliable computers, or computers without which the system using Hermes wouldn't work in any case.

3.2.4 Membership

The membership module uses the gossip module to disseminate information on the addresses of various peers, which is what allows Hermes to operate in the first place. It should probably always be active.

There are no usage requirements for this module. Create a membership context based on a Gossip context, start it, and everything else will happen automatically.

A Membership snapshot can be deserialized on another node, which provides a quick way to initialize the address list of new nodes. Alternately, the connect function of the Core module may be used to gain the initial address information required for the gossip function to operate.

Security

Membership transmits only addressing information. Since this is not considered sensitive information, Gossip makes it impossible to fake, and erroneous information would merely result in an error in the connection negotiation phase, there should be no security considerations.

Assumptions

It is assumed that nodes join and leave the network rather infrequently, which allows membership or address changes to take up to several minutes to propagate through the network using the gossip protocol without incurring unacceptable amounts of downtime. It is possible to reduce this delay by adjusting the gossip configuration, but it is impossible to guarantee a maximum delay, and so there is no way to guarantee that no online nodes have stale address information at any given time.

A node crashing/shutting down and then rejoining the network without changing address does not generate any such membership change, and there is no need to wait for the event to propagate. There is, in fact, no event to propagate.

However, when a node joins the network at a new/updated address it may immediately send and receive messages without waiting for propagation, in the particular case that it starts every conversation. That is, a newly connected node A may talk to node B, and B may then reply. However, if A talks to B which talks to C, then C will not necessarily know the location of A and messages from C to A may fail.

If this is insufficient for a particular application, it is suggested that address information is added to the messages or a central "gossip" server is implemented.

3.2.5 RPC

The RPC module, as its name suggests, allows convenient two-way communication in a function call style.

Functions, which must be of the type $a \rightarrow IO\ b$, are discriminated based on parameter and return type as well as an arbitrary unicode name (String). Individual calls are additionally tagged with a sufficiently large random number in order to prevent any confusion in the event of parallel calls.

In the event that a call is made to a non-registered function, or one of the wrong type, an error is returned. Client calls will block until an answer or error is returned, or a timeout is reached.

On the RPC server side individual calls to any given function are executed in serial, but when there are multiple registered functions they otherwise execute in parallel with each other.

The RPC module is not actually used by Skynet, and is provided mainly as an example.

Future

It should be relatively easy to make parallel/serial execution of requests conditional on user desires, or limit the number or concurrent tasks to a fixed number.

Additionally, it should be possible to add a yieldUntilIdle call to GHC. This would delay execution until one or more OS threads in the runtime is idle, which could be used to dynamically limit the number of processes based on actual load. IRC talks with the GHC developers suggest that this is a probable addition sometime within the next five years.

Superficially, this could also be implemented using forkOS and pthread_setschedprio or pthread_self. However, doing so would break assumptions in the GHC runtime and drastically decrease performance, so is to be avoided.

The use of a string to identify functions may have been a mistake, as there is no guarantee of uniqueness. Instead, future versions will likely use a phantom type for the purpose, with a request that the user define it next to the function being called.

At one point, there was an OpenGL visualizer for Hermes traffic, based on mirroring every message sent to a central computer. This worked pretty well at first, but became unusably cluttered as traffic increased, and eventually bit-rotted and was removed. A similar visualizer, displaying statistical functions of the traffic instead of individual messages (unless you ask it for individual messages) would be of great usefulness.

3.3 AES

The AES package is a Haskell binding to the C-based AES library written by Brian Gladman.[6]

Every effort has been made to maximize performance and prevent any usage possibilities that could cause undefined behavior. At the moment, it is single-copy: The only memory allocation that happens is for the new, encrypted or decrypted bytestring.

There should be no possible usage patterns that cause undefined behavior.

AES offers several levels of abstraction. The main documentation can be found at `http://hackage.haskell.org/package/AES`, so this text will give brief overviews and usage guidelines.

Codec.Crypto.AES provides a pure interface for encrypting or decrypting bytestrings. Both strict and lazy bytestrings are supported. Using the latter, an arbitrarily large bytestring can be en/decrypted using a constant amount of memory.

Unfortunately, for lazy bytestrings, reading N bytes of the output bytestring may require more than N bytes of the input, and indeed more than N plus the AES block size of 16 bytes. Specifically, lazy bytestrings are handled by unpacking them to their component strict bytestrings, which are then handled individually. As such, only complete bytestring chunks are handled.

As it is desirable for encryption output to be independent of bytestring internals, a future version will likely split these at AES block size boundaries instead. This will significantly alter laziness characteristics from what is described above, as overflow from a non-aligned chunk will be prepended to the next chunk, in effect increasing buffering from one to two chunks.

The lazy pure interface is therefore best suited for applications where partial output is not required. You should consider the laziness an optimization, and not depend on output being available before all the input is.

The strict pure interface does not suffer from this unpredictability, but as it does not return an updated initialization vector using it repeatedly will require inventing new IVs each time.

Codec.Crypto.AES.Monad provides a monad transformer built on ST or IO, which allows for fine control of operation order while ensuring safety.

The main motivation for its existence is to interleave operations from different libraries. For example, given a list of (strict) bytestrings, the hash update operation from the corresponding SHA module can be interleaved with the encryption operation, finally encrypting and appending the hash value in the same encryption context.

Incremental encryption/decryption results are available, both inside the monad (as return values from crypt) and in the final lazy bytestring.

Codec.Crypto.AES.IO is the lowest-level binding to the AES library. It provides a slightly abstracted interface, adding garbage-collection and wrapping the C interface with one that uses strict bytestrings.

Every effort has been made to ensure this interface is as close to foolproof as possible. Its major downside is that it can only be accessed from the IO monad.

Codec.Crypto.AES.Random provides three varieties of random-number generation: True random bytes (from /dev/random), cryptographic pseudo-random bytes from an AES cipher in counter-mode, and a System.Random wrapper for the latter to facilitate generating high-quality random values of arbitrary types.

3.4 SHA2

The SHA2 package is a Haskell binding to the C-based SHA library written by Aaron Gifford.[7]

Although Gifford's library supports both SHA-1 and SHA-2, only the SHA-2 functionality is exposed. This is mainly due to a lack of need for SHA-1; since it has been mostly broken[8], its use in new systems is discouraged.

The same design concepts as for the AES binding were used. In this case, the binding is truly zero-copy: No memory is allocated beyond that needed for the SHA context.

However, unlike the AES binding it may be possible to corrupt the runtime by use of unsafe functions in the IO module. For that reason, be very careful if using a function marked unsafe.

SHA2 offers several levels of abstraction. The main documentation can be found at `http://hackage.haskell.org/package/SHA2`, so this text will give only brief overviews and usage guidelines.

In general, all hash functions in this package will accept both strict and lazy bytestrings. The output hash is provided as a bytestring; for convenience, a function to convert a bytestring to a hexadecimal string is provided and exported from each module.

Codec.Digest.SHA provides a pure interface to SHA2, as well as an HMAC function.

Codec.Digest.SHA.Monad provides a monadic interface to SHA2, similar to that for AES.

As explained in the corresponding AES section, the purpose of this interface is to safely allow interleaving of AES and SHA operations. However, the underlying library is missing a function to extract a hash value from an intermediate state (feeding more data into the hash function afterwards), which limits the usefulness of the interface until this has been remedied.

Codec.Digest.SHA.IO provides a low-level binding to the SHA library, adding garbage collection and wrapping it in a bytestring interface.

Similarly to the AES equivalent, all functions require calling from the IO monad. However, unlike that interface, this one can be misused.

As implied above, after a SHA2 context has been finalized it is undefined behavior to continue using it. However, there is no logic in the binding to prevent you from doing so.

The most probable fix is to make the finalization function work on a copy of the SHA state.

Chapter 4

Evaluation

4.1 Design choices

If I could start the design over again, there are a number of things I would do differently.[1]

The choice of keying data by pathname instead of i-nodes has caused a great deal of grief. It was originally done to simplify the system, but between the loss of hardlinks and the increased complexity of moves, it has greatly outlived its usefulness.

Another questionable decision was the decision to create a network API mimicking the POSIX filesystem API instead of creating something very different such as, let's say, a key-value store. Though in this case it would have to be translated on both ends, the simplified network model would lend itself to creating generic functionality for things such as two-phase transactions, which are currently done on a per-function basis.

Additionally, Ceph[5] is another filesystem with largely duplicate functionality, which I was not aware of at the time I started this project. As Ceph's design is superior to mine in most ways, examining it closely for ideas would have been useful. There is still sufficient divergence for a useful project; most notably, Ceph is not meant to be completely maintenance-free.

4.2 Functionality

The system essentially does what it is meant to do, which is to say: Store files. As the commands in the installation/usage chapter are the sum total commands required to set the system up and use it, it has definitely achieved

[1] Starting, perhaps unsurprisingly, with the decision not to make more offline backups.

the "ease of use" requirement, and using it as storage for a variety of common software, including Firefox, has revealed no damaging deviations from POSIX.

There is some less-common software that causes trouble, however. Most notably, the Darcs and Git revision-control systems use hard-links for, respectively, its patch caches and its installed program components. Since Skynet does not support hardlinks, this degrades the performance of Darcs on the system, while Git ends up taking twenty times as much disk space to install as usual.[2]

There was also some functionality on the wish-list that did not make it into the final system. Specifically:

Amazon S2 support would have allowed the system to handle out-of-disk-space conditions rather more gracefully, by pushing files out to Amazon. It could also provide a significant performance boost for Internet access by users that lack a high-bandwidth home connection.

Stashing support - making a frequently-disconnected node, for example a laptop, automatically fetch updates to some subset of the filesystem whenever possible; in effect, acting as a pull-only slave node. As this would immediately lead to users trying to write to stashed files/directories as well, the complexity involved became too much to fit.

Network coordinates - generating some notion of network speed between nodes, and trying to move files to storage nodes close to where the clients that use them are. This turned out to border on AI in difficulty.

Lost node recovery - currently, when files that exist on a crashed node are accessed, the node will be entirely removed from the node list for that file. If the node then comes back up, the filesystem does not understand that there is a near-past version of the file there that could be used if the node is again used as a slave. Ideally, the node would be preferred for this purpose over others, and an rsync algorithm would be used to update the file instead of a straight write.

Maintenance automation works relatively well, with lost replicas being regenerated the quickly after someone tries to access the file/directory. However, since uncommonly accessed files may never trigger this, a periodic process that explicitly checks for node failures and regenerates every resource using that node as a slave would increase reliability. This would require

[2]Still not very much.

implementing lost node recovery first, however, to avoid having to entirely rewrite the contents of crashed nodes when they recover.

4.3 Performance

Performance measurement of Skynet is problematic. Because of the aggressive caching and buffering of uncontested files, most of the time the speed of the system doesn't enter into it; file reads take place at wire speed in all observed cases, while writes complete instantaneously until the write buffer is filled, at which point they are limited to wire speed as the buffer is emptied in a FIFO fashion.

In one dramatic, if improbable example, unpacking and then compiling a Linux kernel reliably took slightly less time on a wifi-mounted Skynet filesystem than on the local disk, as in this case Skynet devolved into an in-memory filesystem from the viewpoint of the compiler. This did however require the available buffer space to be increased to half a gigabyte from the default of 100 megabytes; below this, the unpacking procedure would block as the buffer was filled. It is also likely that the Linux kernel would normally prefer not to use 70% of all available memory as cache.

Where files are contested, performance drastically degrades as every operation becomes a remote procedure call, requiring a round-trip. In this scenario speed is dependent on the latency and bandwidth characteristics of the underlying network; as most calls involve only one node, it has followed the simple (but extremely noisy) equation of $delay = latency + transferSize/bandwidth$.

In either scenario, the cost of encrypting everything is noticeable. On a 100Mbit network a modern laptop was still capable of transferring at wire speed, but at the cost of roughly half its CPU power. Disabling encryption cut that number by nine tenths. As newer Intel CPUs have built-in hardware specifically for AES that has been shown to be highly effective, there would be a great deal of benefit to be had from exploiting these.

The one unquestionably slow operation is directory renames. Since these require the invoking program to fall back to copying the data through the network twice, it is in many cases unusably slow. Some solution is *required*, but the best solution for this particular problem - using traditional, path-less i-nodes - would require a dramatic redesign.

4.4 Final Status

Broken. A slight exaggeration, but unfortunately only a slight one.

Remember the Btrfs experiment? Every file that had recently been written to was corrupted, which unfortunately included the source code of Skynet. There were backups; those were overwritten and also corrupted before the situation was discovered, as their object-file equivalents remained intact. Well, until GHC attempted to recompile them.

Peeling back further layers of backups and restoring as possible, the current status is:

Hermes is undamaged, and fully functional. Being dependencies, so are the AES and SHA bindings. This means Haskell now has a secure message-passing middleware package, with gossip support, which has gathered some acclaim.

Skynet is essentially nonfunctional. Since making it work was the entire point of the thesis, this is somewhat unfortunate. Itemizing:

The resolver is functional. However, some glitch in the storage layer is causing the root node to disclaim responsibility for the root directory, which effectively breaks it.

File creation and writing is functional.

File reading is mostly functional, but the FUSE interface is broken and thus in practice nonfunctional.

File metadata queries/updates are functional.

Directory queries/updates are functional. Directory creation is not, though only by a hair.

Master/slave redundancy is partially implemented, but nonfunctional.

Automatic balancing is nonfunctional.

Failure recovery is nonfunctional.

The rest of this document was written prior to this event, or has been written based on memories of how it worked.

Chapter 5

Conclusion

The project meant to create a distributed fault-tolerant, maintenance-free filesystem, with supporting Haskell message-passing middleware.

Overall, this has succeeded. Although there are plenty of design warts, for at least one use case - bulk media file storage - it works perfectly well, accomplishing the goal of acting as a trouble-free filesystem. Power-cycling tests show that even with a hard machine crash, the filesystem recovers from loss of nodes easily and is capable of reusing them when they come back up.

Last-minute disasters aside, I am quite satisfied with the results, and wouldn't mind using the system myself. However, if I had the free time there are a number of improvements possible; deep design changes that were not obviously a good idea in advance, but in retrospect could have been predicted by asking myself why other unix filesystems do it that way.

The choice of Haskell as an implementation language was a great help. While it is difficult to see the language being used for fast prototyping, for any project of size I believe it would help greatly. The type-checker in particular has saved me probably weeks or months of time, by discovering nine-tenths of the bugs that were ever discovered.

Many of the higher-level code patterns such as transactions, buffers and caches, though not exactly extractable as functions, should be possible to create a generic combinator library for in Haskell. However, this would probably be an equivalently large project on its own, no matter how eventually useful.

Appendix A

Installation and usage

Bearing the status section in mind, installing Skynet or one of its subordinate libraries is relatively easy (and still works). The standard Cabal Haskell packaging system is used, which will automatically download and install all dependencies (including Hermes, AES and SHA; there is no need to use the bundled copies of those, as they have been uploaded to Hackage), with one exception.

The current HFuse version (0.2.2) has a bug (misfeature?), which causes it to change to the root directory at startup; this breaks the rest of the system. To avoid this, a modified version ("HFuse-99.2.2") has been supplied in the package, which should be installed first.

Additionally, Skynet requires the 6.12.1 version of GHC. Previous versions will not work, and 6.12.2 has several bugs that affect its proper functioning. Future Haskell Platform versions of GHC will likely also work; 6.12.1 is the current one.

If you have ip (of the iproute2 package) installed, and a single public or private non-loopback IP, Skynet will automatically bind to this IP at startup. If you don't, you will have to provide the desired IP manually.

Once Skynet has installed, it provides its own usage instructions, which are printed to standard output when Skynet is executed without parameters. Skynet expects to use its current working directory for storage, so a typical workflow would be as follows:

1. For each node, create a storage/state directory, perhaps /.skynet, then cd to that directory.

2. On the administrative node, run skynet create-network. Then, run skynet authorize-node, once for each other node on the network. Copy the resulting authorization files to these nodes.

3. On each node, run skynet create-node in their storage/state directory to initialize it.

4. Finally, to start the system run "skynet run [mount-point]" on each node, starting with the master.

To explicitly make a particular node a slave for a given file/resource, run "skynet enslave [file]" on the node in question, in its state-directory. This will use a hidden file "/.skynetcomm" on the filesystem to order the node. This communication channel currently has no other uses.

Bibliography

[1] The SSH Filesystem on FUSE, http://fuse.sourceforge.net/sshfs.html

[2] Design and Implementation of the Sun Network Filesystem, Russel Sandberg and David Goldberg and Steve Kleiman and Dan Walsh and Bob Lyon, 1985

[3] An overview of the andrew file system, Howard, J.H. and others, Proceedings of the USENIX Winter Technical Conference, 23–26, 1988,

[4] The Coda Distributed File System, Braam, P. J., Linux Journal, edition 50, June 1998, http://www.cs.cmu.edu/afs/cs/project/coda-www/ResearchWebPages/docdir/lj98.pdf, http://www.coda.cs.cmu.edu/

[5] Ceph: Reliable, Scalable, Scalable, and High-Performance Distributed Storage, Sage Weil doctoral dissertation, University of California, Santa Cruz, http://ceph.newdream.net/weil-thesis.pdf, http://ceph.newdream.net/

[6] Brian Gladman, AES implementation, http://gladman.plushost.co.uk/oldsite/AES/

[7] Aaron Gifford, SHA2 implementation, http://www.aarongifford.com/computers/sha.html

[8] Xiaoyun Wang, Yiquin Lisa Yin, Finding Collisions in the Full SHA-1, 2005, http://people.csail.mit.edu/yiqun/SHA1AttackProceedingVersion.pdf

[9] Dominique A. Heger, Workload Dependent Performance Evaluation of the Btrfs and ZFS Filesystems, DHTechnologies, Austin, Texas http://www.dhtusa.com/media/IOPerf_CMG09DHT.pdf,

[10] https://btrfs.wiki.kernel.org/index.php/Main_Page, btrfs on irc.freenode.org,